Activations

Christophe Alias

Objectifs

- On veut compiler les fonctions
- Les paramètres, les variables locales et la valeur de retour sont des entiers
- Les paramètres sont passés par valeur

```
int fact(int n) {
  int tmp;
  if(n == 0) return 1;
  else {
    tmp = fact(n-1);
    return n*tmp;
} }
```

Langage d'entrée

- Appel de fonction: E +::= id(E, …, E)
- Return: S +::= return E;
- Définition: F ::= int id(int id, …, int id) {D S … S}
- Nouvel axiome: P ::= F … F

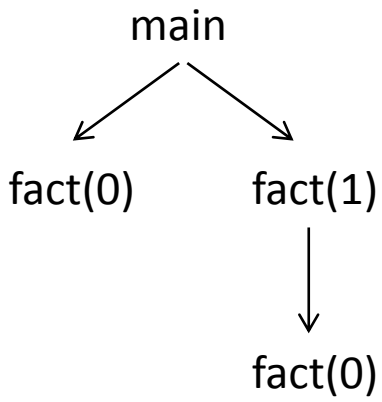
```
int fact(int n) {
  int tmp;
  if(n == 0) return 1;
  else {
    tmp = fact(n-1);
    return n*tmp;
} }
```

Activations

- Chaque appel de fonction est une activation
- Pour chaque activation, il faut stocker les paramètres, les variables locales et l'adresse de retour.
- On les place dans un enregistrement d'activation

```
int fact(int n) {
  int tmp;
  if(n == 0) return 1;
  else {
    tmp = fact(n-1);
    return n*tmp;
}}

int main()
{ return fact(0) + fact(1); }
```

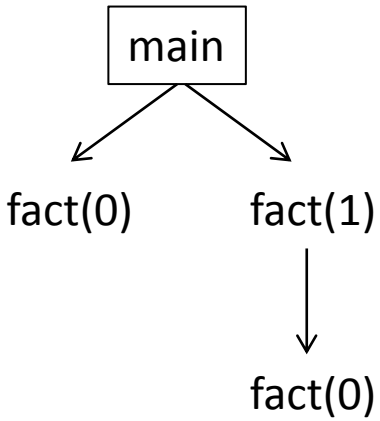


Activations

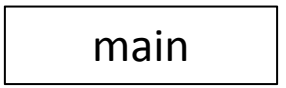
- Chaque appel de fonction est une activation
- Pour chaque activation, il faut stocker les paramètres, les variables locales et l'adresse de retour.
- On les place dans un enregistrement d'activation

```
int fact(int n) {
  int tmp;
  if(n == 0) return 1;
  else {
    tmp = fact(n-1);
    return n*tmp;
}}

int main()
{ return fact(0) + fact(1); }
```



Activations

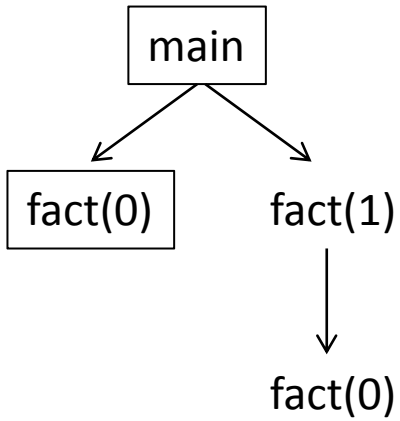


Activations

- Chaque appel de fonction est une activation
- Pour chaque activation, il faut stocker les paramètres, les variables locales et l'adresse de retour.
- On les place dans un enregistrement d'activation

```
int fact(int n) {
  int tmp;
  if(n == 0) return 1;
  else {
    tmp = fact(n-1);
    return n*tmp;
}}

int main()
{ return fact(0) + fact(1); }
```



Activations

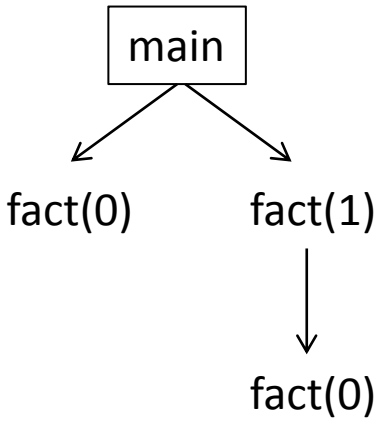
main
fact(0)

Activations

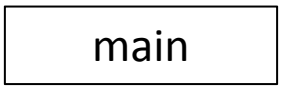
- Chaque appel de fonction est une activation
- Pour chaque activation, il faut stocker les paramètres, les variables locales et l'adresse de retour.
- On les place dans un enregistrement d'activation

```
int fact(int n) {
  int tmp;
  if(n == 0) return 1;
  else {
    tmp = fact(n-1);
    return n*tmp;
}}

int main()
{ return fact(0) + fact(1); }
```



Activations

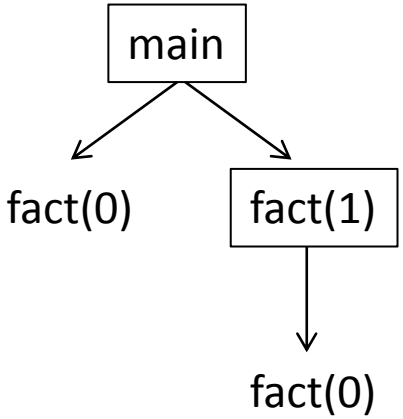


Activations

- Chaque appel de fonction est une activation
- Pour chaque activation, il faut stocker les paramètres, les variables locales et l'adresse de retour.
- On les place dans un enregistrement d'activation

```
int fact(int n) {
  int tmp;
  if(n == 0) return 1;
  else {
    tmp = fact(n-1);
    return n*tmp;
}}

int main()
{ return fact(0) + fact(1); }
```



Activations

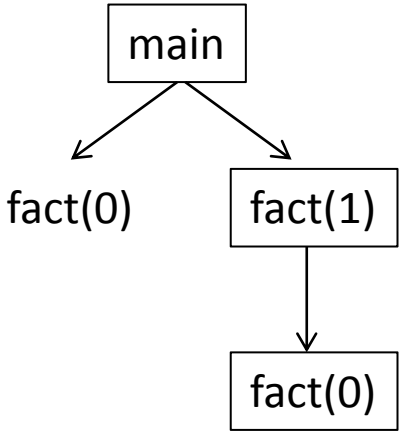
main
fact(1)

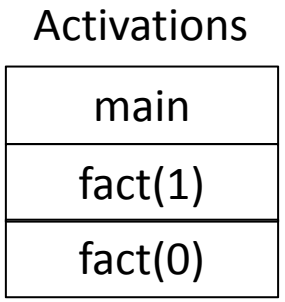
Activations

- Chaque appel de fonction est une activation
- Pour chaque activation, il faut stocker les paramètres, les variables locales et l'adresse de retour.
- On les place dans un enregistrement d'activation

```
int fact(int n) {
  int tmp;
  if(n == 0) return 1;
  else {
    tmp = fact(n-1);
    return n*tmp;
}}

int main()
{ return fact(0) + fact(1); }
```



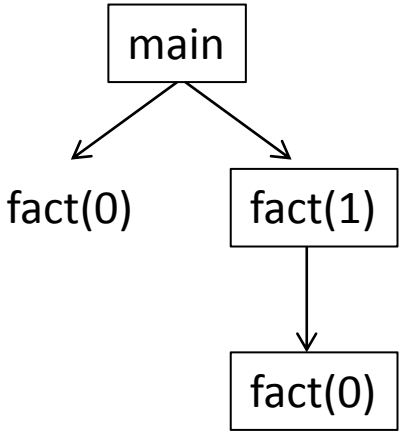


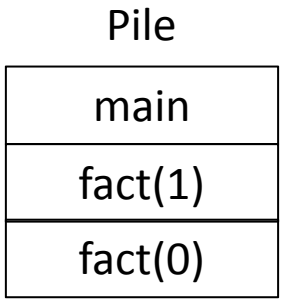
Activations

- Chaque appel de fonction est une activation
- Pour chaque activation, il faut stocker les paramètres, les variables locales et l'adresse de retour.
- On les place sur une pile dans un enregistrement d'activation

```
int fact(int n) {
  int tmp;
  if(n == 0) return 1;
  else {
    tmp = fact(n-1);
    return n*tmp;
}}

int main()
{ return fact(0) + fact(1); }
```



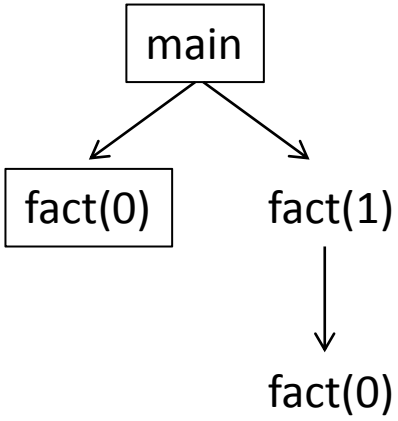


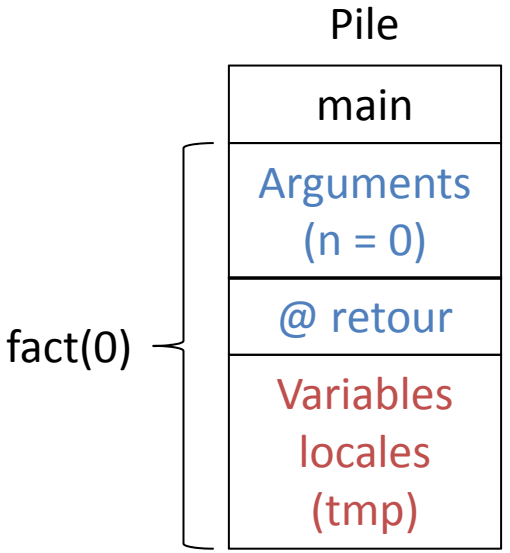
Conventions d'appel

- L'appelant enregistre les arguments et l'adresse de retour
- L'appelé alloue les variables locales

```
int fact(int n) {
  int tmp;
  if(n == 0) return 1;
  else {
    tmp = fact(n-1);
    return n*tmp;
}}

int main()
{ return fact(0) + fact(1); }
```





En pratique…

- La convention d'appel est imposée par le système d'exploitation:
  - Passage d'arguments (pile, registre+pile, ordre)
  - Retour du résultat (registre, pile)
  - Qui sauvegarde/restaure les registres
  - Qui désalloue les arguments
- Conventions dominantes:
  - Convention System V AMD 64 ABI (Linux 64)
  - Convention Microsoft x86-64 (Windows 64)

Convention Linux 64

- Passage d'arguments
  - Premiers arguments dans les registres:
    $rdi, $rsi, $rdx, $rcx, $r8, $r9 (entiers)
    $xmm0–7 (flottants)
  - Arguments restants sur la pile (*right-to-left*)

printf("Résultat: %d",n);

```
mov     rdi,fmt_char    ; format for printf
mov     rsi,[rbp+16]    ; first parameter for printf
mov     rax,0           ; no xmm registers
call    printf          ; Call C function
```

Convention Linux 64

- Passage d'arguments
  - Premiers arguments dans les registres:
    $rdi, $rsi, $rdx, $rcx, $r8, $r9 (entiers)
    $xmm0–7 (flottants)
  - Arguments restants sur la pile (*right-to-left*)
- Retour du résultat
  - Entier: $rdx:$rax
  - Flottant: $xmm1:$xmm0

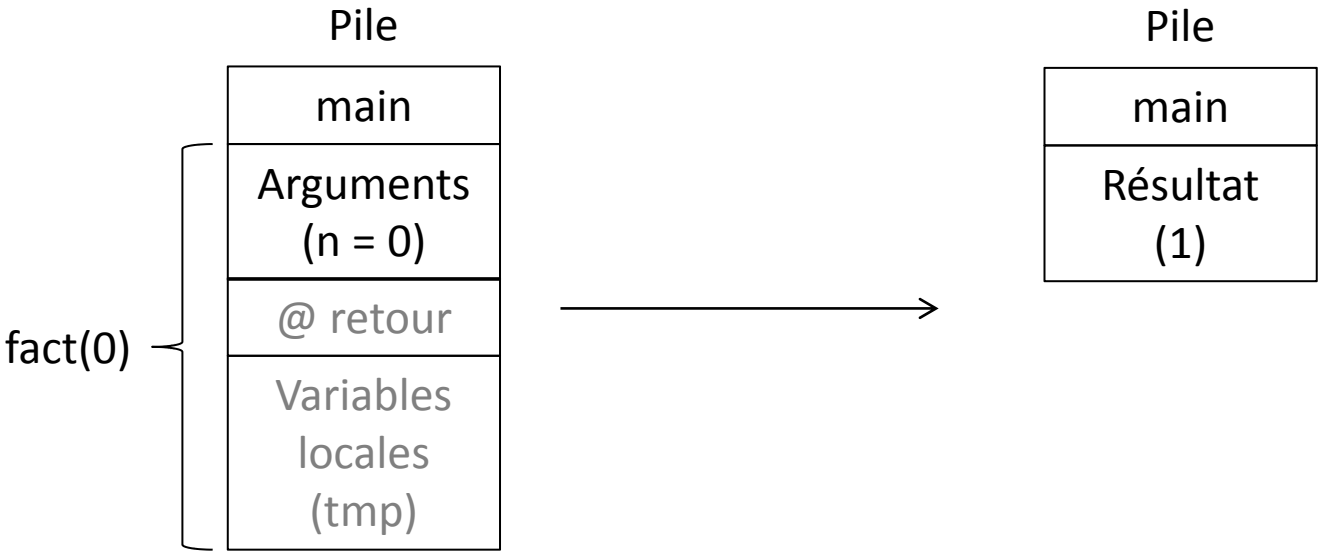
```
mov     rdi,[rbp+16]
call    malloc
; rax is a pointer to the allocated space
```

Convention Windows 64

- Passage d'arguments
  - Premiers arguments dans les registres:
    $rcx, $rdx, $r8, $r9 (entiers)
    $xmm0–3 (flottants)
  - Arguments restants sur la pile (*right-to-left*)
- Retour du résultat
  - Entier: $rdx:$rax
  - Flottant: $xmm1:$xmm0

Convention Java VM

- Passage d'arguments
  - Arguments sur la pile (*left-to-right*)
- Retour du résultat
  - Sommet de la pile



Extension de la machine à pile

call @

empiler(@retour)

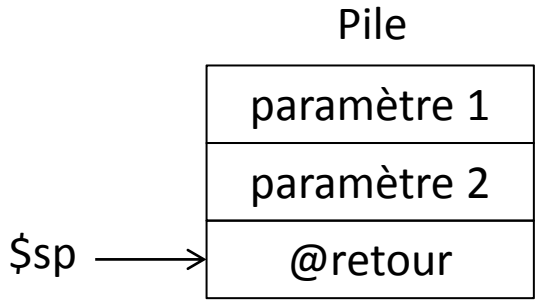
saute à @

ret n

dépiler(@retour)

$sp = $sp + n

saute à @retour



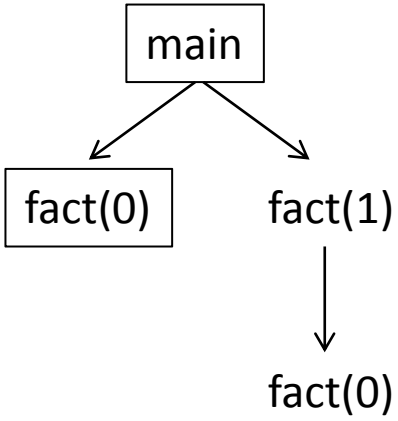
Traduction d'un appel de fonction

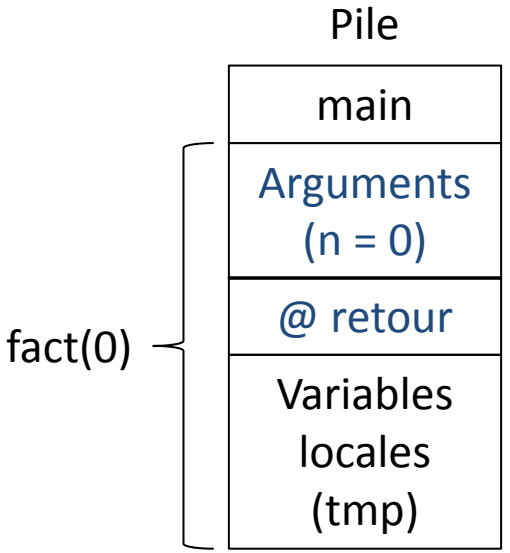
E +::= id(E, …, E)

⟦id(E1, …, En)⟧ = ⟦E1⟧ … ⟦En⟧ call id

```
int fact(int n) {
  int tmp;
  if(n == 0) return 1;
  else {
    tmp = fact(n-1);
    return n*tmp;
}}

int main()
{ return fact(0) + fact(1); }
```



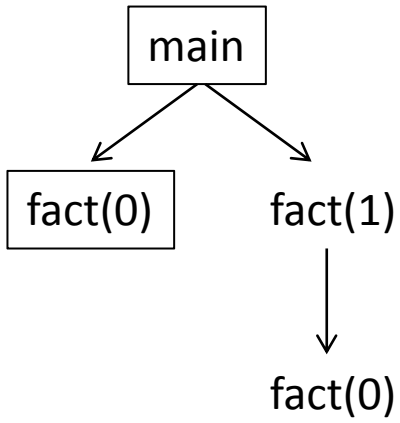


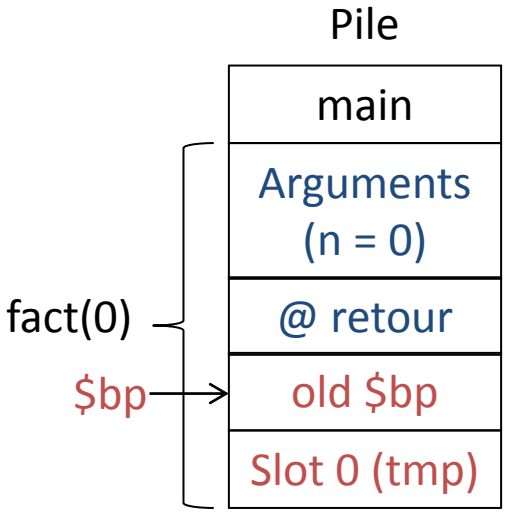
Traduction d'une fonction

F ::= int id(int id, …, int id) { D S … S }

⟦int id(int id1, …, int idp) {
    int id'1, …, id'q;
    S1 … Sr }⟧ =

id:
  alloc q
  ⟦S1⟧ … ⟦Sr⟧
  free
  ret p





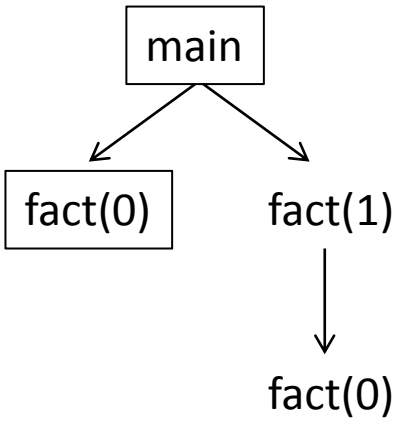
Quizz (1)

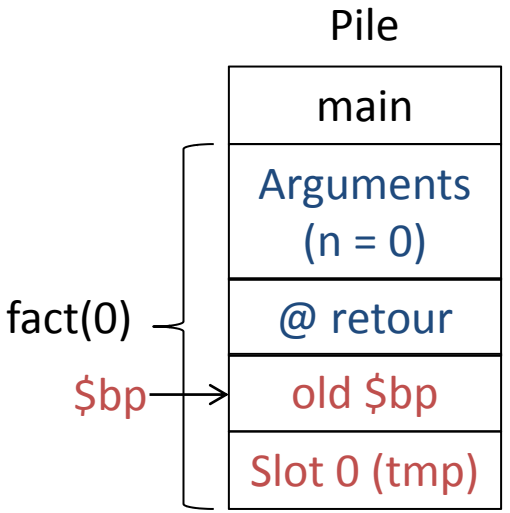
Donner:

- δ(n), δ(tmp)
- les adresses de n et tmp en fonction de $bp

```
int fact(int n) {
  int tmp;
  if(n == 0) return 1;
  else {
    tmp = fact(n-1);
    return n*tmp;
}}

int main()
{ return fact(0) + fact(1); }
```



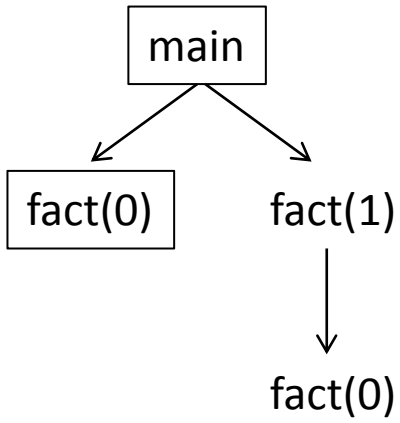


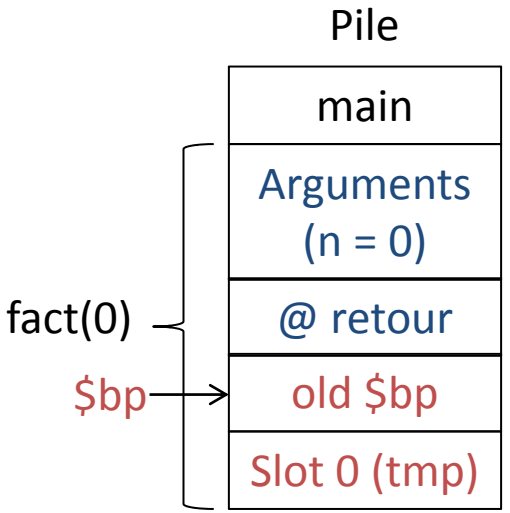
Quizz (2)

Que fait le code suivant
(dans le corps de fact):

fact:

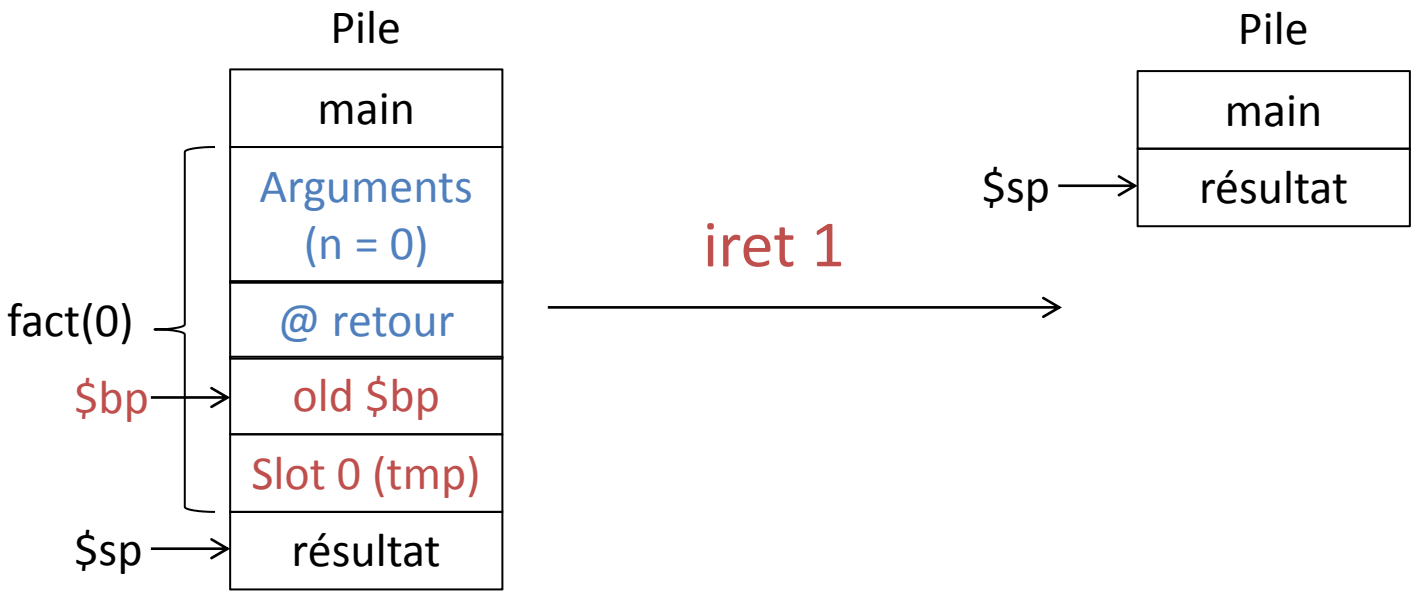
```
...
push -3
mpush 0
teste
...
```





Traduction du return

iret n
  dépiler(résultat)
  désallouer l'enregistrement d'activation
    (dont n arguments)
  empiler(résultat)
  sauter à @retour



Traduction du return

S +::= return E;

⟦return E;⟧ = ⟦E⟧ iret nb_parametres

Traduction du programme

P ::= F … F

⟦F1 … Fp⟧ =

```
call main
mov rax,0
ret 0
⟦F1⟧
…
⟦Fp⟧
```